

*Virtual Functions, Pure Functions &
Abstract Base Classes*

Casting

We saw the following:

```
Base *R = new Derived();    // okay
```

To use the member functions of **Derived**, one has to “remind” the compiler that we know that the object **R** is pointing to is a **Derived** object:

```
Derived *Q = static_cast<Derived*> (R);
```

Overriding

The derived class can provide its own version of a function that appears in the base class.

(Note that a function is determined by its name and the types of its arguments (and *not* its return type).)

This raises the question of: which version of a function is executed.

Virtual and Nonvirtual

If a function is labeled **virtual** in the **Base** class, then the **Derived** version of a function is always used.

Otherwise, which version is used is determined by how the object is referenced. (We omit the details.)

Comment on Virtual Functions

Note that if a function is labeled **virtual** in the **Base** class, it is automatically **virtual** in the **Derived** class.

In Java all functions are implicitly **virtual**

Destructors

Destructors should be virtual: the **Derived** version should always be run on a **Derived** object.

Pure Functions

A **virtual** function might have no implementation at all in the **Base** class. This is indicated by writing

```
virtual void scream( ) = 0;
```

in the class definition.

Abstract Base Classes

An **abstract base class** is one with at least one pure function. We cannot instantiate an object of an abstract base class.

(Java: An **interface** is a class without member variables all of whose member functions are pure.)